SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract., **10**, 37–57 (1998)

Research

Traceability Aspects of Impact Analysis in Object-oriented Systems

MIKAEL LINDVALL[1]* and KRISTIAN SANDAHL[2]

[1]*Department of Computer and Information Science, Linkoping University, S-581 83 Linkoping, Sweden*
[2]*Ericsson Radio Systems, AB, Application Centre for Mobile Switching and Radio Networks, Box 1248, S-581 12 Linkoping, Sweden*

SUMMARY

Impact analysis is an essential activity as cost and effort estimation is based on its outcome. Impact analysis, as performed in an industrial object-oriented system, has been analysed and evaluated in this case study. Preceding the design phase of the fourth release of the PMR system (PMR R4), the consequences of a set of new requirements were analysed. Traceability links were established between each new requirement and the objects predicted to be changed in the design object model, which represent the C++ classes in the system. The impact analysis as performed in this study was successful in the sense that the software engineers in the study prefer this way of working. When asked, they claim they are sure the number of objects to be changed is correct, while they are less sure about the number of man-hours required. The analysis of the impact analysis shows that individual objects are often predicted almost correctly, but that the number of actually changed objects is often greater than the number of predicted objects. The study also shows that tracing by interviewing knowledgeable software engineers is far more common than consulting object models and other kinds of documentation. A factor that prohibits software engineers using object models more extensively is lack of detailed information. We see a promising potential for using traceability, together with the use of existing object models, to achieve a more structured impact analysis process and accurate prediction.

KEY WORDS: impact analysis; traceability; evaluation; object-orientation; case-study; industrial project

1. INTRODUCTION

Being able to predict cost and effort for development projects is essential in any field of engineering, and software engineering is no exception. A typical situation occurs when

* Correspondence to: M. Lindvall, Department of Computer and Information Science, Linkoping University, S-581 83 Linkoping, Sweden. E-mail: mikli@ida.liu.se

Contract grant sponsor: Ericsson Radio Systems AB
Contract grant sponsor: National Board for Industrial and Technical Development; Contract grant number: 9303280-2

CCC 1040–550X/98/010037–21$17.50

Received 3 July 1996
Revised 28 September 1997

an existing software system is about to undergo major change because of new requirements. The stakeholders of the system (e.g., users, customers, product managers) often present a variety of new requirements: new functionality to be deployed, errors to be corrected and performance improvements to be carried out. When limited resources (e.g., man-hours, calendar time) are at hand or when the day for delivery should be negotiated, it is important to be able to predict the cost for each individual requirement. In our practical work, cost estimation is preceded by the task of assessing the effects of making a set of changes to a software system. This task is called impact analysis (Queille *et al.*, 1994).

The impact analysis results in a list of the requirements together with the predicted impact, in terms of software entities of the system that have to be changed, for each requirement. Accurate impact analysis is, however, very hard to accomplish and is in part dependent on long experience and on learning from mistakes in the past. In a joint research project with Ericsson Radio Systems (ERA) we have had the opportunity to conduct a case study of an object-oriented industrial project, Performance Management Traffic Recording (PMR) developed in Objectory (Jacobson *et al.*, 1992). In order to describe how impact analysis is performed by professional software engineers, we have observed, documented and analysed the outcome of the impact analysis conducted in the PMR-project, release 4 (PMR R4). The purpose of this paper is:

- to present a theoretical framework for impact analysis based on observations from the first release of PMR;
- to account for how the impact analysis approach is conducted in practice;
- to collect and analyse software change metrics related to impact analysis; and
- to establish a baseline by evaluating the effectiveness of the current impact analysis approach.

The study object, the PMR project, is characterized in the Appendix in order to provide the reader with information about the circumstances under which the data were collected. This helps in judging the transferability of our results to other projects.

2. THEORETICAL FRAMEWORK

Impact analysis

There are many interpretations of the term ‘impact analysis’ (Arnold and Bohner, 1993). Here we mean the activity of predicting the cost of implementing a new requirement in terms of the impact on the source code, by the use of any available information about the system. It is, of course, important to estimate the impact on all system-related documents, as mentioned by Turver and Munro (1994). In this study, abstract design models are used as a tool for, mainly, documentation of the results from the impact analysis, whereas impact on other related documents, such as user manuals, are not included. Impact analysis can be conducted on different granularity levels. In an object-oriented system, the object level, i.e., predicting the objects that have to be changed, can be chosen as the appropriate granularity. As an object is defined by its operations (i.e.,

methods), attributes and associations with other objects (Jacobson *et al*., 1992), it is also possible to achieve a higher level of detail, for example, also predicting the methods that have to be changed. The level of granularity chosen should be a subject for cost–benefit analysis. In this study we have limited the evaluation to the object level, deeper analysis is left for future work.

To be able to conduct impact analysis as accurately as possible, all possible and available ways have to be examined. Our experience is that knowledge about the existing system is considered as the most important and widely used source of information in practice (Lindvall, 1995). Such knowledge can be documented in different ways. Designers and programmers who have been working on the system for a long time generally have a very clear picture of how the system works and of how different parts depend on others. It is often perceived as more efficient to ask the person who initially wrote a specific piece of software about how it should be changed to account for a new requirement. The source code is also considered a good source for information and is often consulted for detailed information.

Discussing possible changes with the designers and programmers, which we call the *interview method*, is attractive when time is short, but is not completely satisfactory:

- Undocumented knowledge will be lost if key designers are no longer available.
- The interview method is not necessarily systematic, important consequences might be accidently overlooked.
- Dependencies between different parts of a system, even in relatively small ones, are often hard to understand without performing extensive analyses of the system.

Prediction

Impact analysis is about predicting the impact on the system as an effect of a new requirement, an activity conducted prior to the implementation of the requirement. When the requirement is eventually implemented, it is possible to compare the prediction with the actual outcome to evaluate the effectiveness of the impact analysis approach in use. The software objects can be divided into four sets. Regard all objects in the system as a set, named ‘system’, and assume that a subset of objects in the system are predicted to be changed, then these objects constitute the estimated impact set (Arnold and Bohner, 1993). At the same time it is implicitly predicted that the other objects will remain unchanged. This makes two sets: objects predicted to be changed and objects predicted to remain unchanged.

When the new requirements are implemented, the objects can again be divided into two sets: actually changed objects, which constitute the actual impact set, and unchanged objects.

When comparing the prediction with the actual outcome, four sets are formed (Figure 1):

- Unchanged objects being predicted as unchanged.
- Changed objects being predicted as changed.

These two sets represent situations where the prediction was correct. Below are the two situations when prediction fails:

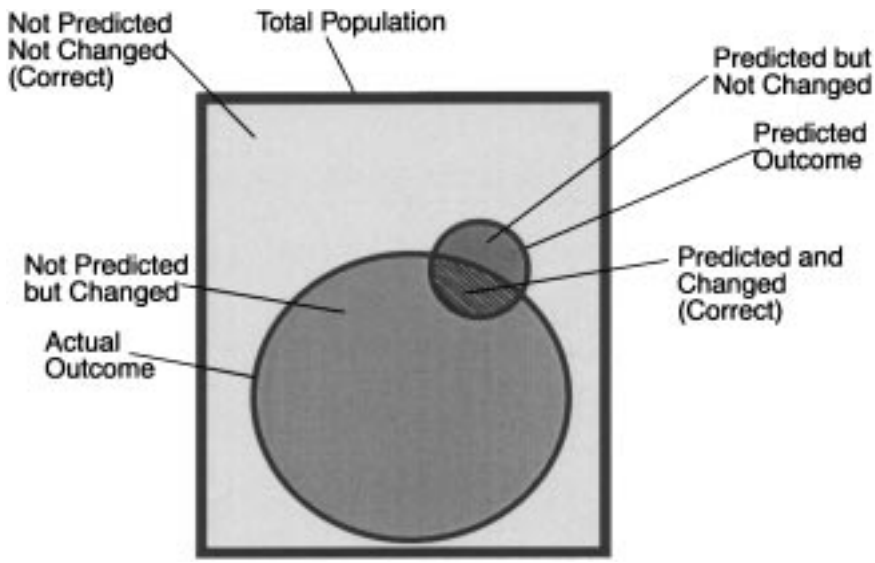

Figure 1. General diagram showing how prediction versus actual outcome is related

- Unchanged objects being predicted as changed.
- Changed objects being predicted as unchanged.

Traceability

Impact analysis involves tracing software objects which are candidates for change, possibly represented in various abstract design models. This makes traceability closely related to impact analysis. General traceability is the ability to trace from one abstract model to another, where requirements traceability is the ability to trace from a requirement to, for example, its implementation in source code. Traceability is fundamental to life cycle meta-models such as Basili's *iterative reuse model* (Basili, 1990). In Basili's model, software includes not only the source code, but also up-front documents such as requirements—and design specifications, which are regarded as *models* at various abstraction levels of the software (Jacobson *et al.*, 1992). Maintenance, in Basili's perspective, initially performs and documents changes in requirements which are subsequently propagated through the analysis and design models to the source code. Understanding how change propagates from model to model is made easier if the system possesses a high level of traceability. Proponents of this approach, such as Pfleeger and Bohner (1990) and Turver and Munro (1994), assume a high level of traceability, which in practice implies that:

- all models of the software are consistently updated;
- it is possible to trace dependent items *within a model (vertical traceability)*; and
- it is possible to trace corresponding items *between different models (horizontal traceability*).

Traceability can be expressed using directed graphs consisting of nodes and edges. Each requirement, each design component and each part of code is denoted by a node and every link of dependence is denoted by an edge. The graph of dependencies resembles a web and in Figure 2 it is depicted as a general traceability model showing how different models correspond to each other. Literature on traceability often presents the subject on a principle level. To complement these contributions we performed a traceability 'experiment' between different models to gain better program understanding in the first release of the PMR system, see Lindvall and Sandahl (1996), and classified different forms of tracing as follows.

(1) Tracing via *explicit links*, which are often part of a development environment. An example is the Objectory SE (Objective Systems SF AB, 1993b) which provides a technical means for explicit traceability, but leaves the user with the decision as to how to use them.
(2) Tracing by using *references*, which are defined as textual references to different documents.
(3) *Name tracing* assumes a consistent naming strategy and is used when building models. It is performed by searching items with names similar to the ones in the starting model.
(4) *System knowledge* and *domain knowledge* can be used by an experienced software developer by tracing concepts using his knowledge about how different items are interrelated.

The goal of all four classes is the same, but the cost and reliability differ. Maintaining links can be tedious, but the cost of system experts is very high.

3. INVESTIGATION METHOD

In order to study, analyse and report on maintenance of an industrial scale object-oriented system, we have observed the development and maintenance of PMR, a system for recording and analysing telecommunication traffic events, since the very outset in late

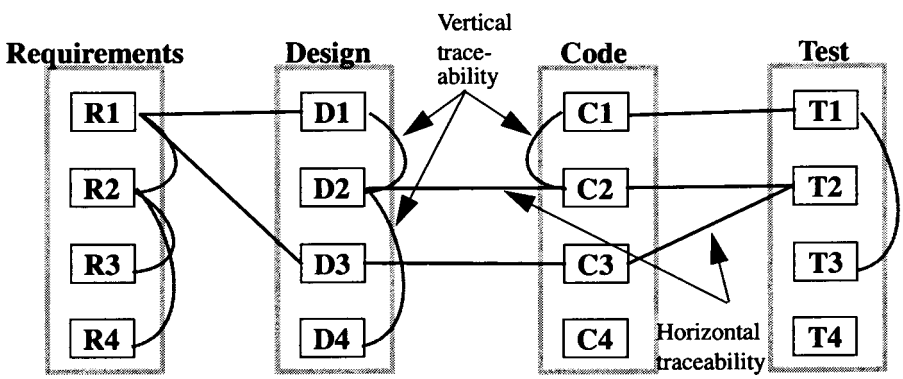

Figure 2. The general traceability model as a graphical web, idea from Pfleeger and Bohner (1990)

1992 to the completion of the sixth release in February 1996. The focus of our study was directed at issues of object-oriented software maintenance and especially at the task of understanding and improving impact analysis. Owing to this long term *participatory observation method* (Gummesson, 1988) we had the opportunity to become acquainted, on both a general and a detailed level, with the developers, the process and the product by participating in project meetings and by analysing the available documents and other information. To complement the observations we have furthermore conducted structured interviews with developers and technical managers. Preceding the design phase in release four of the PMR-project (PMR R4), a regular cost analysis based on impact analysis was conducted, which we have observed and analysed. In this section we discuss the goals we wanted to achieve, the information available and the methods used. Generally speaking, the goals were to be able to describe and analyse the impact analysis process as conducted in PMR R4 and evaluate the impact analysis approach in use by comparing the predicted outcome with the actual one. The process is described in terms of

- input to the process,
- the observed process, and
- output from the process.

Process input

Input to the process of impact analysis in PMR R4 included both information about the new requirements and information available about the previous release (PMR R3). The new requirements were described in ordinary language and categorized in terms of functional improvements, errors reported or performance improvements. Examples of information about the previous release include: requirements, use case models, object models, source code and traceability information. Such information is denoted *design information* in this paper, and can, theoretically, be used when tracing candidates for change. Most of the design information was stored and retrieved using the Objectory SE tool. We were interested in describing the information available for performing impact analysis. This was achieved by analyses of design information, interviews and project meeting participation. Some information was retrieved via a conventional relational database (RDB) operating on the content of the Objectory SE database.

Process

As the impact analysis for the requirements under consideration was distributed amongst several software engineers, we decided to document the process in different ways. We started by letting one of the software engineers fill in a form for each requirement on which he conducted impact analysis. During an interview after the impact analysis, the software engineer explained the circumstances for every filled field. Based on these forms and the interview, we derived a strategy and structure for interviews with other software engineers who (a) had been involved in the impact analysis of the requirements that we covered in detail, and (b) who conducted impact analysis on any other requirement. The interviews served as a basis for the description of the observed process.

Process output

The output from the process of impact analysis involves the predicted outcome per requirement in terms of the names of predicted software objects. We collected this information to be able to compare it with the actual outcome. When all new requirements had been implemented, it was possible to compare the predicted outcome with the actual one. Hence, we captured information about which software objects were actually changed, and why, according to what requirement, an object was changed. This was achieved by letting programmers trace back, from those software objects in the source code that had been changed, to requirements. The result from back-tracing was analysed and verified with each programmer.

Comparison

The predicted outcome was compared with the actual outcome. We identified the weaknesses of the prediction process by comparing the software objects predicted to be changed with the software objects actually changed.

4. OBSERVATIONS

In this section observations are presented in the order of input, process and output. Input is described by the new requirements and the existing design information. Process is described mainly by the information used and how tracing was performed. Output is described by how the result of the impact analysis is documented.

Characterization of new requirements

The new requirements that were subjects for impact analysis and implemented during the PMR R4 project can be divided into three categories:

- functional improvements, F1–F14;
- performance enhancements, P1–P3;
- error corrections based on trouble reports, E1–E4.

The 14 functional improvements can be traced back (Finkelstein, 1991) both to responses from end users when using earlier releases of the product and from in-house users when testing the system. The functional improvements included in this release are mainly those that aim at an increased usability, rather than more functionality. This means that the functional improvements in general are related to the user interface and how the user interface should be changed to comply with the users' work flow. The functional improvements are specific and described on a detailed level that could generally not have been specified earlier in the evolution process, but only after substantial use of the product. However, the two most extensive functional improvements (F2 and F3) are of a different nature. F2 concerns porting the system to a new operating system; F3 concerns new functionality central for the user.

To improve performance, an in-depth investigation of the system running in a variety of circumstances was conducted. Tools for measuring where execution time was spent in the system have been used extensively in order to trace bottlenecks. The investigation resulted in three different measures taken (P1–P3), each targeted at a specific bottleneck in the system, and are included as any other requirement.

The use of earlier releases of the product resulted in four trouble reports and concrete measures taken as a result of the impact analysis. These measures are included in the PMR R4 project as any other requirement. Impact analysis has been applied in different ways for the required changes. The most common way was to trace design objects using available information sources, as discussed further below. F2, however, was in part analysed using a specific porting tool which scanned the source code and identified source code statements where a change had to be made.

Available design information

The available design information consists of requirements, use case models, object models, source code, modification information and traceability information. The information was captured mainly in tools, such as Objectory and SCCS (Rochkind, 1975), and also in ordinary C++ source code files (Stroustrup, 1986). We will discuss the content and the status of the information.

The *Objectory* method (Objective Systems SF AB, 1993a; Jacobson *et al.*, 1992) together with its accompanying CASE-tool *Objectory SE* (Objective Systems SF AB, 1993b) was used for storing requirements, and for analysis and design in the PMR-project. The methodology as well as the tool were used *per se* without mixing them with other methods. The PMR system was partly implemented in C++ and some parts were implemented using Sybase, a relational database (Codd, 1970). The entire system was modelled in Objectory because the team required a model of the system in all its aspects regardless of the implementation technique.

The models developed during the PMR R1 project, i.e., object models and use case models, have been kept and maintained as much as time pressure has allowed with the intention of supporting future maintenance. Hence, the models have been extended to reflect the enhancements for PMR R2, PMR R3 and PMR R4. Updating the design model according to actual implementation has been performed. The design object model is thus an accurate representation of the implementation. Applied to the C++ part of the system it means that almost every C++ class has a corresponding design object with the same name (some of the C++ classes do not have a corresponding design object, as these classes are considered as representing too low an abstraction level). We say there is a one-to-one mapping from design objects to classes. Regarding more detailed information, such as member functions/methods and relations between entities, the discrepancy between the design object model and the C++ source is larger (Lindvall, 1997).

The *use case model* mirrors the high-level requirements for the system and has not been altered since the development of the first release (R1).

Each source code file in the PMR project comprises one and only one C++ class and SCCS is used for version management of each individual file. Whenever a programmer is about to modify a C++ class, the corresponding source file first has to be checked out

from SCCS, the modification performed and then the file checked in again. Thus, SCCS provides a complete log over all modifications performed to all files in the project. Every modification thus has a notification about who made it, date and time, and a comment provided by the programmer describing the modification. To assess detailed information about the modification it is possible to run SCCS Diff, which compares two files and shows exactly what has been modified.

Traceability information, in the form of traceability links, was provided from the original 14 requirements to the nine use cases in the use case model. Between each use case and its participating design objects there were also explicit links, which made it possible to answer such questions as: how is this use case realized in terms of design objects? See Lindvall (1994) for more detailed descriptions. Because of the fact that the majority of the design objects had a corresponding C++ class with the same name as the object, we claim that name tracing was possible between the design object model and the source code. By using the information provided by SCCS it was possible to achieve traceability between requirements and C++ classes on the modification level.

The observed impact analysis process

We describe, on a high level, actions taken immediately before the impact analysis process, the input, the output and its documentation and the impact analysis process itself.

Product managers provided a set of suggested changes based on comments from various stakeholders. The set was divided into three priority levels, which served as input to the requirements selection process. New requirements on the two highest priority levels became subjects for impact analysis and were stored in the requirements model in Objectory SE.

The general process description prescribed that each new requirement should be traced to the design objects that caused the change. As the new requirements were functionally orientated, this was a function-to-object approach, but owing to the limited size of each new requirement, they were not considered as use cases, as would normally be the case in Objectory. The analysis was documented by introducing traceability links between each requirement and the changed design objects. The result was a complete set of requirements of changes for PMR R4, where each requirement was linked via traceability links to the set of design objects predicted to be changed.

The most widely used tracing technique was the use of *system knowledge* and *domain knowledge* which are parts of the *interview method*. The interview method means that the software engineer responsible for conducting impact analysis for a particular requirement interviews his colleagues on the issue and the design objects that might be changed. The conclusions were examined thoroughly and verified during group meetings where the result of the impact analysis for each requirement was discussed in detail among all developers. The main reason behind the extended use of the interview method is an unverified belief among software engineers in the project that it takes considerably longer to consult documentation than to ask a knowledgeable person. One software engineer estimates that 90–95% of the changed design objects can be found by using the interview method. Another reason behind this behaviour is that software engineers do not trust documentation and therefore are reluctant to use it. They often complain about the gap between object models and the actual source code and the lack of confidence in the

documentation. When their system knowledge is not sufficient and the interview method does not help, the next alternative is to consult the source code. By using Sniff (TakeFive Software, 1995), a source browser for C++, it is possible to select a C++ class and to trace dependencies based on message passing to other classes.

A possible explanation for the lack of confidence in the documentation, even though it was of good quality on the object level, is the fact that not all information needed for impact analysis was provided by the models. This forced the software engineer to examine the source code. This is due both to the fact that models in some cases were not up-to-date (accidental lack of information), and also that not all relevant data are intended to be stored in any of the models (intended lack of information). One example of the latter is the fact that dependencies between objects were captured in the object model, but not dependencies between methods. When we described the approach of the *forest of impact* (Queille *et al.*, 1994), which means that dependencies between objects are used for identification of candidates for change, which in turn further generates new candidates for change, the software engineer said he worked according to the approach, but not in a systematic way.

Traceability links were implemented between requirements for R1 and the original use cases, but could not be used in this case since the original use cases were defined so broadly that the new requirements fitted into the use cases without any necessary change. Traceability was accomplished informally, without documenting how the new requirements fitted into the existing use cases in Objectory. Given a direct question, however, the technical project leader could do the mapping easily. One suggested use of traceability links is when requirements change and it should be possible to trace where in the system the requirement is implemented. In this case, however, the usefulness of the existing traceability links was limited because of the difference in granularity between old and new requirements. As stated above, *new* traceability links were introduced during the impact analysis process between each new requirement and the predicted design objects.

Process output

The result from the impact analysis was the complete set of new requirements for PMR R4, where each requirement was linked via traceability links to the set of predicted design objects in the design object model describing the previous release (PMR R3). Each design object had a corresponding C++ class. Hence, the result was an indirect relationship between new requirements causing change in existing software, via the design object model. The traceability links were later used when planning and distributing the design and implementation tasks among software engineers as there was a shift from functionality to objects (classes). By the use of traceability links it was easy to answer the question: 'what requirements cause a change in this object in order to account for the new requirements?'. This information was also used as an instruction on how to transform the existing design for PMR R3 to become PMR R4.

When the impact analysis was completed, the 14 functional improvements, the three performance enhancements and the four error corrections had been analysed using impact analysis. All these are regarded as new requirements, which amounted to 21 new requirements to deal with in the project. The 14 functional improvements were emphasized,

which is why some of the other requirements had no assigned design objects at the time. During the project, these new requirements were implemented, but additional change requests also had to be dealt with, for example trouble reports not registered at the time of project planning. The result from the impact analysis was used in calculating estimated cost for each requirement. These figures were used in negotiations with product management in determining the design assignment.

5. QUANTITATIVE RESULTS

While the previous section concerned observations and qualitative results, this section concerns results from our analysis of change metrics. The results include classifications and measures, mainly of source code related activities and software objects. The section includes a comparison between predicted outcome and actual outcome on the object/class level.

Classification of modifications

By interpreting the comments programmers stored in SCCS together with their new comments from a questionnaire it was possible to classify the underlying reasons for all modifications made in the R4 release project. The reasons are divided into two main groups depending on their predictability. We conclude that the following seven categories are reasons for modification.

New requirements, *predictable:*

- RQ—change according to a previously known new requirement. All modifications performed according to one of the 21 new requirements were collected in this group.
- DR—change according to an existing design rule that was previously not adhered to, for example, resorting member functions in alphabetical order. As design rules generally are documented, it could have been possible to predict this change if knowledge about the current situation and about the design rule had been available.

Additionals, *non-predictable:*

- NRQ—change according to a new requirement, which was not known at the time of the impact analysis and hence not predictable.
- TR—change according to a new trouble report, not predictable.
- Improve—code was changed in order to improve it, for example to produce better trace routines or to improve maintainability. Not predictable, but possibly suppressible by management instructions.
- NDR—change according to a new design rule, for example, revision history is moved from the header of the source code file to be managed entirely by SCCS.
- ?—do not know. It is not possible to determine why this piece of code was changed.

Analysis of requirements

For each class, we measured how many requirements were the reason for the change in that class. This led to Table 1.

The table is to be interpreted as follows. 54% of the classes that had to be changed were changed as an effect of only one requirement, 22% of the classes were changed because of two requirements, 8% were changed because of three requirements and 2% of four or more requirements. It is also worth noticing that 14% of the changed classes were not changed because of any particular requirement. The reason is that they were changed because of non-predictable trouble reports and the like. To create a fair evaluation of predictive capability of impact analysis, these changes should be excluded in further analysis.

We can also see that designers had managed to orthogonalize requirements well: a majority of objects are changed because of one or two requirements only (76%). This results in the relations between requirements and objects in many cases being one-to-many; a requirement causes change in many objects, but one-to-one in the object–requirement direction; an object is often changed because of one single requirement. The data thus speak against the myth that relations between requirements and code are complex (Soloway, 1987).

Analysis of classes

An analysis of the predicted software objects compared with the actual outcome regarding C++ classes produces the data presented in Figure 3. It is important to note that this analysis takes a perspective where the set of all predicted objects/classes is compared with the set of all changed objects/classes for the entire release. This view is useful for an overall analysis of the impact on the system regardless of single requirements.

There are now 136 C++ classes in the system. Nine classes have been added since PMR R3. 30 design objects, each corresponding to one and only one class each, were predicted as needing to be changed because of one or more new requirements. Of these 136 classes, 120 have a corresponding design object and thus were predictable.

The actual result is that all of the 30 predicted classes were also changed because of one or more new requirements (correct prediction), which means that according to

Table 1. Originating requirements per class

Number of originating requirements	Reason for change	Proportion of modified classes
0	One of the following: DR, NRQ, TR, Improve, NDR or ?	14%
1	RQ	54%
2	RQ	22%
3	RQ	8%
4 or more	RQ	2%

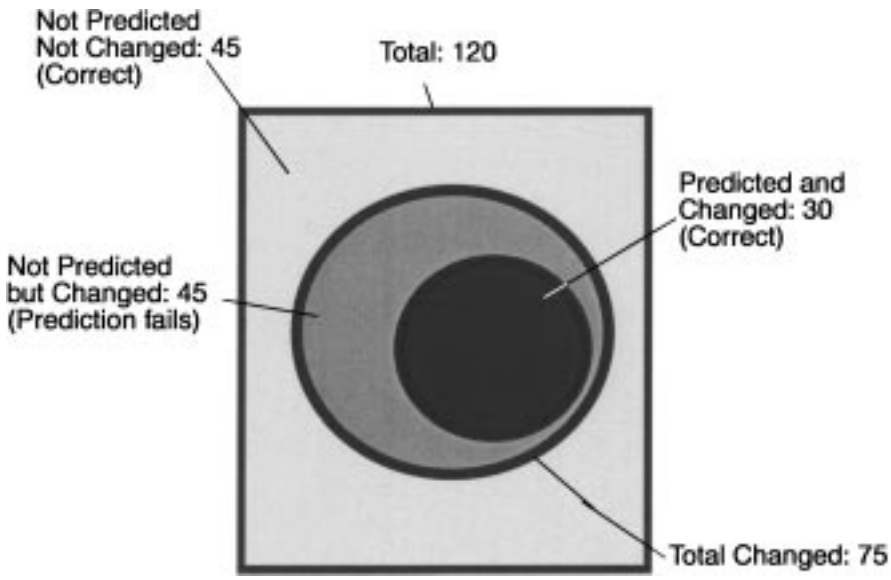

Figure 3. The relation between prediction and actual outcome in terms of predicted and changed classes analysed on the system level based on data from PMR R4

this perspective no classes were predicted to be changed and actually not changed (prediction fails).

45 classes, all with corresponding design objects, other than the predicted 30 were also changed.

In total, 75 classes of the 120 were changed.

45 classes were implicitly not predicted for change and were not subject to change (correct, but implicit prediction).

The analysis still remains at the release level as the system has been analysed in terms of how the *set* of new requirements was predicted to cause change in the system compared with the actual outcome. How *each* requirement in the set causes change in the system is discussed below.

Comparison

Now we can analyse how each requirement was predicted to cause change of the software objects and compare that with the actual outcome. The comparison made here comprises only the C++ related parts of the system. In Figure 4 the prediction and the outcome is presented. In Table 2 the complete set of figures is presented. It is important to note that the selection of software objects is based on the following criteria.

- Only those software objects that have both a design object and a C++ class associated with them are included. By this selection, all classes that were changed but do not have a corresponding design object, and hence are not predictable, are omitted.
- Only those software objects that were changed because of a new requirement are

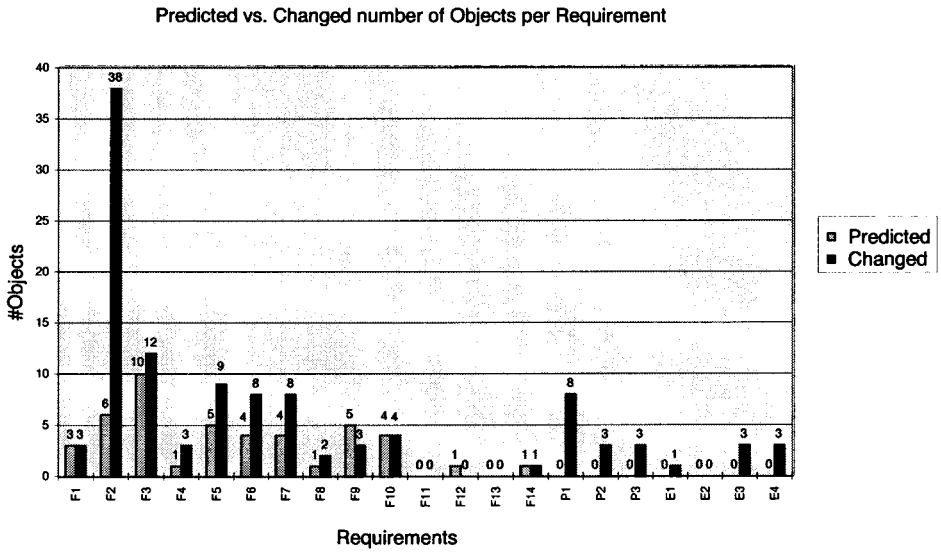

Figure 4. How new requirements were predicted to impact objects (left bar) versus the actual outcome in terms of changed objects (right bar). Type of new requirements from left to right: functional improvements, performance enhancements and error corrections. Note that some requirements, e.g., F11, were planned and implemented by changes in other parts of the system, but not in C++ code

Table 2. Predicted versus actual number of changed classes per requirement

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	P1	P2	P3	E1	E2	E3	E4
Predicted outcome	3	6	10	1	5	4	4	1	5	4	0	1	0	1	0	0	0	0	0	0	0
Predicted but not changed	1	1	3	0	2	0	1	0	2	0	0	1	0	0	0	0	0	0	0	0	0
Predicted and changed	2	5	7	1	3	4	3	1	3	4	0	0	0	1	0	0	0	0	0	0	0
Changed but not predicted	1	33	5	2	6	4	5	1	0	0	0	0	0	0	8	3	3	1	0	3	3
Actual outcome	3	38	12	3	9	8	8	2	3	4	0	0	0	1	8	3	3	1	0	3	3

included. By this selection process, all classes that were changed because of some additional change are omitted. See Table 1 for more details.

Table 2 presents the functional improvements (F1–F14), the performance enhancements (P1–P3) and the error corrections to account for trouble reports (E1–E4). Please note that it is not possible to summarize the columns in the table in order to reach the previous

figures as the change of perspective (from system to requirements) leads to different calculation results. As an example: a class predicted to account for requirement A only, but is actually changed to account for requirement B only, will end up as *predicted but not changed* and *changed but not predicted* in the current perspective, but will end up as *predicted and changed* in the release-level perspective.

Table 2 shows the comparison on a relatively detailed level. The top row 'predicted outcome' shows, for each requirement, how many software objects were predicted to be changed to meet the requirement. This can be compared with the bottom row 'actual outcome' which shows, for each new requirement, how many classes were actually changed to implement the requirement. Let us discuss two extremes.

- The new requirement F1 was predicted to cause change in three software objects, and three were changed.
- The new requirement F2 was predicted to cause change in six software objects, but 38 were changed.

These figures are, however, too coarse-grained for deeper analysis. If we analyse F1, for example, we still do not know if the prediction was close to the actual outcome or not. It only tells us about the predicted number, not about whether the right classes were predicted or not. Compare with a situation where a medical test shows positive results for three patients, A, B and C, when the actual case is that patients D, E and F have infections. The predicted number of infections was right, but the actual individual patients involved were wrong. Rows two to four in the table provide more information: for F1, one software object was predicted but actually not changed, and one software object was not found during impact analysis but was actually changed. This leaves two software objects that were both predicted and changed.

The common pattern is that

- too few classes were predicted for change; and
- that some individual classes were wrongly predicted.

We will now analyse the data along two dimensions: *the number of predicted classes versus the number of changed classes* and *correctness in terms of the overlap between the set of predicted classes and the set of changed classes*. The first measure is selected because the number of predicted classes is often used to estimate the effort and cost of the new requirements. The second measure is selected because of the importance of identifying a correct set of classes to be changed.

Only the first nine new requirements are selected for the analysis. Performance enhancements and error corrections are omitted because of their difference in nature and because emphasis during impact analysis was on functional improvements. Requirement F2 is also omitted due to its different nature, porting to a new operating system. The remaining nine functional improvements are relatively homogenous in nature. We begin by presenting, for each requirement, how the prediction of the number of changed classes relates to the actual number of changed classes. See Table 3.

Table 3. The relation between predicted and actual number of classes per functional requirement

F1	F3	F4	F5	F6	F7	F8	F9	F10
100%	83%	33%	55%	50%	50%	50%	166%	100%

The difference is expressed as a percentage and calculated by dividing the predicted number by the actual number of changed software objects. The measure can be interpreted as an answer to the question: 'what is the relation between the number of predicted software objects and the number of actually changed software objects?'. According to Table 2 the new requirement F1 was predicted to cause change in three software objects and three were actually changed. The grade of predicting number of classes is 100% (3/3). For the new requirement F3, 10 software objects were predicted for change and 12 were actually changed. The grade of predicting number of classes is 83% (10/12).

The following can be read from Table 3.

- The predicted number was higher than the actual number in one case (F9).
- It was exactly right in two cases (F1 and F10).
- The predicted number was lower than the actual number in six cases (F3, F4, F5, F6, F7 and F8).
- The predicted number was half, or less than half, of the actual number in four cases (F4, F6, F7 and F8).

Our conclusion is that there is a tendency to underestimate the number of changed classes.

The next analysis deals with the comparison on the individual level, i.e., what is the relation between the predicted individual classes and the actually changed classes. See Table 4.

Correctness is expressed in the form of a percentage and is calculated by dividing the number of individual classes that were predicted for change and also changed by the number of individual classes predicted for change. This provides a measure that can be interpreted as 'how many of the classes that were predicted to be changed were also actually changed?'. According to Table 2 the new requirement F1 was predicted to cause change in three software objects and three were actually changed, but, of the individual classes in the predicted set of software objects, only two were actually changed. Correctness is 67% (2/3). For the new requirement F3, 10 software objects were predicted and 12

Table 4. Correctness: the relation between the predicted individual classes and the actual outcome

F1	F3	F4	F5	F6	F7	F8	F9	F10
67%	70%	100%	60%	100%	75%	100%	60%	100%

were actually changed, but, of the individual classes in the predicted set of software objects, only seven were actually changed. Correctness is 70% (7/10).

The following can be read from Table 4.

- The predicted individual classes were 100% correct in four cases (F4, F6, F8 and F10).
- The lowest correctness rating in predicting individual classes was 60% (F5 and F9).

Our conclusion is that the impact analysis approach in use results in estimated impact sets that, with relatively high probability, contain objects that will be changed.

6. SUMMARY AND CONCLUSION

It is clear that in a setting where many knowledgeable and experienced software engineers work, it is regarded easier to trace objects that need to be changed by interviewing experts rather than consulting documentation. We refer to this as tracing using domain and system knowledge, and the interview method. Compared with the theoretical view, the documentation, the models and traceability information would be the natural way of proceeding. It has been shown in experiments that it is possible to trace this kind of information if traceability information is available (Abbattsista *et al*., 1994; Epping and Lott, 1994), but it has not been verified in an industrial-scale setting. The result of this study does not show that tracing using documentation, models and accompanying tools is a preferred way of solving the task of impact analysis. This is surprising since much effort is actually spent on keeping documentation up to date. Instead, the study indicates that exerienced software engineers use implicit models, i.e., they have the models in their mind, and that tracing by interviewing and using Objectory by connecting predicted design objects with the requirement under consideration is a successful way of documenting the result. This makes it easy to change from a functional to an object-oriented approach when proceeding from impact analysis to design. Work in the design phase is based on the parts predicted to be changed rather than on requirements. This orthogonal change in approach makes it easier for each software engineer to control his classes and decreases the need for file control, which is needed if many software engineers are able to edit the same files.

Impact analysis as performed in this study was successful in the sense that the software engineers interviewed prefer this way of working and that they are confident about the correctness of the result. When asked, they say that they are sure the number of changed classes is correct, while they are less sure about the number of man-hours required. The evaluation of the impact analysis shows that individual classes are often predicted almost correctly, but that the number of actually changed software objects is often greater than that predicted for change, even when uncommon types of requirements are removed. This result is at odds with the confidence the software engineers show in the result. Our conclusion is that it would be of great value if the impact analysis process could be improved. Today it is well defined when and with what purpose impact analysis is performed, but only little is said about how the analysis should be performed. Our contribution provides a method for evaluating the impact analysis, which is an instrument

in assessing the effect of future improvements. The evaluation process itself can also be improved by prompting the developers for the reason of change as files are checked out.

For the future, it is desirable to decrease the work load and dependence on the experts and be able to produce more accurate predictions. This requires a different, more active, use of models, documentation and tools. As long there is a lack of necessary information in object models, we should not assume that they will be used for impact analysis other than as a basis for high-level discussions about the system. We have seen a promising potential for using traceability information together with existing object models and the Objectory tool to achieve a more structured process for impact analysis and accurate prediction, even if there is much work left to accomplish before project developers will accept such an approach.

APPENDIX. DESCRIPTION OF THE PMR PROJECT

The PMR project was developed by Ericsson Radio Systems (ERA) which develops the software for systems for mobile telecommunication. At a very high level we recognize three important components of a mobile telecommunication system, namely mobile station, network, and monitoring and administration system. The *mobile station* (MS) is the wireles telephone itself. The *network* consists of a set of interconnected AXE exchanges (MSC and BSC) and performs radio communication with each MS via a particular cell. The *monitoring and administration system* comprises a platform with several optional applications. The section at ERA that was studied in this research is responsible for the development of one particular application, namely, performance management traffic recording (PMR).

The operator of the network needs to record traffic events during radio communication, which is the purpose of PMR. It is possible to set a particular MS in focus and to record its interactions with different cells, or to set a particular cell in focus and record communicating MSs. A recording consists of *measurements* which contain information about the actual transmitting status in the network. Measurements are created at intervals of 720ms and hold information on the serving cell and the neighbouring cells. The recording process is initiated by the operator, who specifies the host exchange in the network (the orderer), the type of recording, the subject for recording and the time window. The result of the recording is collected by several exchanges (collectors), sent to PMR, parsed, analysed and stored in a database and later used by the operator to trace malfunctions in the system.

The PMR project fulfilled its primary goals and was conducted within calculated cost and time, as the outcome from the project contained the functionality specified in the requirements specification at an acceptable quality level. A family of products was developed by incremental development and several versions were based on the same analysis model and different design models.

The *Objectory* method and its accompanying CASE-tool *Objectory SE* were used for analysis and design in the project. The method was used without intervention of other methods, which makes the results from this study relevant for a broad group of Objectory users. Some documents, not supported by Objectory, were produced by other tools, for

example, detailed descriptions of the user interface and documents closely related to the implementation.

The input to Objectory consisted of the *requirements specification*, which described functional and non-functional requirements. Only the functional requirements were explicitly modelled in Objectory, as it does not explicitly support non-functional requirements modelling. A rough description of requirements analysis in Objectory is to organize the functional requirements into use cases. The developers responsible for writing the requirements specification used use case in the requirements specification to denote a high-level functional requirement. At the same time naming conventions of use cases were maintained. This resulted in a straightforward transition from requirements to use cases.

The output from Objectory consisted mainly of detailed design descriptions comprising an extensive design object model of the structure of the system, an overall use case description and *interaction diagrams* showing the complex parts of the system. A rudimentary skeleton for C++ was also generated. The Objectory process also produced a number of models preceding the design model. These models were, however, never intended to be passed on to the programming team, but kept for maintenance purposes.

The reason for the models was made clear in the project. The underlying rationale for creating and maintaining various models was future maintenance, since the task for the project was to design a family of products simultaneously. The *domain object model* is intended to serve as a model showing the real world in this particular context and provides a common terminology and description of the important concepts of the problem domain. The *analysis object model* is used as a description of an ideal system showing commonalities and differences among the products in the family. The *design object model* describes each product derived from the analysis model and is a high-level representation of the actual implementation. The various *use case models* mirror the requirements for the system and how they are refined during the development process.

The PMR system was partly implemented in C++ and partly using a relational database (RDB). The development team knew from the start that both C++ and an RDB would be used, but not which parts would be implemented using which technique. Therefore, the entire system was modelled in Objectory and it was decided during the design phase how each part should be implemented.

Most of the developers were not used to Objectory or other object-oriented modelling techniques and an extensive series of courses was organized. To make the transition to the new technology even easier, the technical manager for the project was employed directly from the vendor, Objectory AB. C++ as a programming language was familiar to some of the programmers. The staff represented a broad spectrum of developers with great differences in experience of object-oriented modelling and implementation. Most of the developers were eager to start learning the new technology and held high anticipations. We have strong reasons to believe that for the purpose of our research, the negative learning effects from Objectory were outweighed by the enthusiasm of faithfully applying the method. During the project, evaluations of the method and the tool were performed twice by project management. These evaluations were used to collect and assess the developers' experience and to improve their use of the method and the tool. To summarize, many developers appreciated working with Objectory, but several improvements regarding

the tool were suggested, for example, features for configuration management and a spell checker.

Acknowledgements

The authors are greatly indebted to Benny Odenteg, Göran Petterson and their co-workers on the PMR project. We are also grateful to Al Goerner, University of Missouri, Kansas City for insightful comments on our work and to Ivan Rankin for proof-reading and correcting our English and to three reviewers for comments on the final paper.

References

Abbattsista, F., Lanubile, F., Mastelloni, G. and Vissaggio, G. (1994) 'An experiment on the effect of design recording on impact analysis', in *International Conference on Software Maintenance 1994*, IEEE Computer Society Press, Los Alamitos CA, pp. 253–259.

Arnold, R. S. and Bohner, S. A. (1993) 'Impact analysis—towards a framework for comparison', in *International Conference on Software Maintenance 1993*, IEEE Computer Society Press, Los Alamitos CA, pp. 292–301.

Basili, V. (1990) 'Viewing maintenance as reuse-oriented software development', *IEEE Software*, **7**(1), 19–25.

Codd, E. F. (1970) 'A relational model for large shared databanks', *Communications of the ACM*, **13**(6), 377–387.

Epping, A. and Lott, C. (1994) 'Does software design complexity affect maintenance effort?', in *Proceedings of the 19th Software Engineering Workshop*, Goddard Space Flight Center, Greenbelt MD, 16 pp.

Finkelstein, A. C. W. (1991) 'Tracing back from requirements', in *IEE Colloquium on Tools and Techniques for Maintaining Traceability During Design*, (Digest No. 180), IEE, London, pp. 7/1–7/2.

Gummesson, E. (1988) *Forskare och konsult—om aktionsforskning och fallstudier i foretagsekonomi (Qualitative Methods in Management Research*), Studenlitteratur, Lund, 135 pp.

Jacobson, I., Christersson, M., Jonsson, P. and Overgaard, G. (1992) *Object-oriented Software Engineering*, Addison-Wesley, Menlo Park CA, 524 pp.

Lindvall, M. (1994) 'A study of traceability in object-oriented systems development', Licentiate Thesis, Linkoping Studies in Science and Technology, No 462, Department of Computer and Information Science, Linkoping University, Institute of Technology, Sweden, 149 pp.

Lindvall, M. (1995) 'Traceability aspects of impact analysis in the fourth release of an industrial object-oriented system', Memo 95-03, ASLAB, Department of Computer and Information Science, Linkoping University, S-581 83 Linkoping, Sweden, 10 pp.

Lindvall, M. (1997) 'An empirical study of requirements-driven impact analysis in object-oriented systems evolution', Ph.D. thesis, Linkoping Studies in Science and Technology No.480, Department of Computer and Information Science, Linkoping University, Institute of Technology, Sweden, 252 pp.

Lindvall, M. and Sandahl, K. (1996) 'Practical implications of traceability', *Software: Practice and Experience*, **26**(10), 1161–1180.

Objective Systems SF AB (1993a) *Objectory Analysis and Design 3.2 Process*, Objective Systems SF AB, Kista, Sweden, 185 pp.

Objective Systems SF AB (1993b) *Objectory Analysis and Design 3.2 Tool*, Objective Systems SF AB, Kista, Sweden, 138 pp.

Pfleeger, S. L. and Bohner, S. A. (1990) 'A framework for software maintenance metrics', in *Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos CA, pp. 320–327.

Queille, J., Voidrot, J., Wilde, N. and Munro, M. (1994) 'The impact analysis task in software

maintenance: a model and a case study', in *International Conference on Software Maintenance 1994*, IEEE Computer Society Press, Los Alamitos CA, pp. 234–242.
Rochkind, M. J. (1975) 'The source code control system', *IEEE Transactions on Software Engineering*, **1**(4), 364–370.
Soloway, E. (1987) 'I can't tell what in the code implements what in the specs', in *The Second International Conference on Human–Computer Interaction*, Elsevier Science Publishers B.V., Amsterdam, pp. 317–328.
Stroustrup, B. (1986) 'An overview of C++', *ACM SIGPLAN Notices*, **21**(10), 7–18.
TakeFive Software (1995) *Sniff+ Release 2.0*, sniff-gst-002 edition, TakeFive Software GmbH, Salzburg.
Turver, R. J. and Munro, M. (1994) 'An early impact analysis technique for software maintenance', *Journal of Software Maintenance: Research and Practice*, **6**(1), 35–52.

Authors' biographies:

Mikael Lindvall is currently assistant professor in software engineering, processes and methods, at the Department of Computer and Information Science at Linkoping University, Sweden.

In 1984 he founded and managed a 20 person company in CAD applications programming. Increasing interest in principal software engineering questions based on experiences with programming projects made him also start an academic career.

In 1991 he received an M.S. degree in computer science and engineering. In 1997 he received a Ph.D. degree in computer science and especially software engineering, both from Linkoping University, Sweden.

His Ph.D. thesis was based on a long-term empirical study of an industrial object-oriented project at Ericsson, conducted over several releases. Dr Lindvall's interests are quality aspects of software development methods, software evolution, object-orientation, requirements engineering, impact analysis and traceability.

Dr Lindvall is a member of the program committee for the IEEE Workshop on Empirical Studies of Software Maintenance. E-mail: mikli@ida.liu.se

Kristian Sandahl is a member of ZeLab, the systems engineering research laboratory at Ericsson Radio Systems in Linkoping, Sweden. He also holds a part time position as associate professor at the Department of Computer and Information Science at Linkoping University.

In 1983 he received an M.S. degree in computer science and engineering at Linkoping University, and worked as a consultant in parallel with post-graduate studies and research administration. In 1992 he received a Ph.D. degree in computer science and founded a research group with the goal of performing empirical studies in industrial software and knowledge engineering. Since October 1995 Dr Sandahl has been employed by Ericsson with special responsibility for facilitating co-operation with the academic world.

Dr Sandahl has published empirically founded methods for knowledge-based systems design, innovations for biochemical information systems, and empirical observations from large-scale telecommunication software development. His major interests are knowledge management, knowledge engineering, CSCW, industrial software engineering, quality improvement paradigm, inspection processes and empirical research methods. E-mail: kristian.sandahl@era.ericsson.se

